THE MODEL-DRIVEN DEVELOPMENT APPROACH & FORMAL SPECIFICATION FOR AIR-TRAFFIC CONTROL SYSTEM OPERATIONS: PRACTICAL COMMENTS AND CASE STUDIES

Fabio Seiti Aguchiku[*]
Atech Negócios em Tecnologia S/A
Rua do Rócio, 313 – 2º andar, São Paulo, SP, (11) 3103-4600, FAX (11) 3103-4601.
faguchiku@atech.com.br

Rafael Leme Costa[*]
Atech Negócios em Tecnologia S/A
Rua do Rócio, 313 – 2º andar, São Paulo, SP, (11) 3103-4600, FAX (11) 3103-4601.
rlcosta@atech.com.br

Eric Conrado de Souza
Atech Negócios em Tecnologia S/A
Rua do Rócio, 313 – 2º andar, São Paulo, SP, (11) 3103-4600, FAX (11) 3103-4601.
ecsouza@atech.com.br

Newton Maruyama
Dept. de Engenharia Mecatrônica e Sistemas Mecânicos – Escola Politécnica da Universidade de São Paulo, Av. Prof. Mello Moraes, 2231, São Paulo, SP, (11) 3091-5337.
maruyama@usp.br

ABSTRACT

This note is part of continuing research that aims at introducing model-driven development techniques to system development cycle. Some modeling tools and analysis technics are reviewed and applied to achieve improved productivity for the ATC system development process. These tools represent a large set of resources, ranging from modeling and formal specification languages to software for model-driven development. Case studies are presented for modeling and analyses of various functionalities of a simplified version for a flight plan manager, currently in operation at various air-traffic centers throughout Brazil. Also considered, an application for receiving and processing flight tracks from ADS-B data.

Keywords: Model-Driven Development, Formal Specification, Model Checking, Air-Traffic Control, ADS-B Communication

[*] This note is the partial result of preliminary research into modeling methods being developed for two projects in the masters of engineering program at the Polytechnic School of the University of São Paulo.

1. INTRODUCTION

Software Engineering can be understood as "the development of reliable and high quality software systems on schedule" and within budget constraints, (Ferré, Vegas, 1999), see also (Basili, Caldiera, 1992). Great effort has been dedicated to better characterize what methods should be used to achieve these high-level goals which ultimately define Software Engineering. One possible approach to software engineering is based on the Model-Based Systems Engineering (MBSE) concept. According to INCOSE (International Council on Systems Engineering), MBSE is the "formalized application of modeling to support system requirements, design, analysis, verification and validation activities beginning in the conceptual design phase and continuing throughout development and later life cycle phases", (INCOSE, 2007). In this setting, it is envisioned that system development would evolve from a document- centric activity to a model-centric one. This major shift in terms of processing information during the many phases related to the product life-span would allow for enhanced knowledge capture, improved communication between the various stakeholders, improved ability to manage system complexity and, thus, increase productivity and quality.

Systems and operations in the Air Transportation domain are software-intensive, computationally distributed, human-in-the-loop, with various measures of associated complexity[2] and representing great "potential for accidents arising from unsafe interactions among non-failed components" (Fleming, Leveson, Placke, 2013). Apparently, as Air Traffic Control (ATC) and Air Traffic Management (ATM) systems in particular, and other purpose systems in general, become more dependent on automation with the purpose to secure the realization of future demands, "human controllers will begin to shift from direct control to supervision of automation, which can complicate human decision-making" (Ibid.). Thus, dependability on safety-critical and reliable systems will increase substantially. In the present context, and unless otherwise stated, a system will represent a software system. This is particularly true for software employed in safety-critical systems. Given the expectation that software elements in these, or other specific application, systems must not fail, and that software solution and development become ever more complex, see (Cernosek, Naiburg, 2004), this assumption is even more applicable. Hence the advent of model based approaches to software engineering.

An interesting related work to ATC and code generation from models can be found in (Whittle, Kwan, Saboo, 2005). The authors in this report were able to automatically generate code from scenarios of intended behavior and integrate it with the – then under development – existing CTAS (Center TRACON Automation System) system and tested with satisfactory success. The CTAS system provides automation tools for planning and controlling air traffic arrival. More recently, Carrozza et. al. (2013) have investigated the use of MDE (Model-Driven Engineering) in the development of new generation ATM systems.

The sections that follow present the use of modeling tools employed to specify and model systems. Some case studies are given. This study reflects an initial investigation to assess the feasibility of adopting different methods of software development; and, because of the academic nature of this study, tool utilization herein is made under the premise of a non-commercial license.[3]

The remainder of this paper is organized as follows: a brief overview of related model-based ideas and concepts is given in Section 2. Section 2 is further structured into four subsections: subsection 2.1 briefly introduces the MDA approach, subsection 2.2 reviews the xtUML language used for modeling, executing and transforming models, subsection 2.3 details the modeling case study for the Flight Plan Processing in ATC systems, and the ADS-B modeling case study is mentioned in subsection 2.4. Section 3 presents a discussion of the modeling process

[2] Sometimes defined as intellectual unmanageability.

[3] Different licensing will apply if and when conditions of software use vary.

and verification with two formal specification languages and the corresponding tools, by highlighting their commonalities and comparing their distinct modeling features and functionalities. Sections 4 presents a brief discussion on MDD adoption and alternatives. Final remarks are given in Section 5. Section 6 lists the references used and Section 7 contains a list of Acronyms.

2. MODEL DRIVEN ENGINEERING

In this Section, attention will be given to the system modeling approach eventually applied for verification by model execution. Caprio (2008), from Techtarget.com, defines Model-Driven Development (MDD) as the use of "models to capture high level information, usually expressed informally, and to automate its implementation, either by compiling models to produce executables, or by using them to facilitate the manual development of executables." Markus Völter in (Völter, 2010) underlines the one great advantage attained through model-driven development by arguing that conceptual system architecture and implementation details or technology decisions are easier to evolve during development when decoupled. This goes hand-in-hand with the idea of productivity, as mentioned above with the MBSE approach. The main goal of MDD is explicitly summarized by Atkinson and Kühne, (2003): "The underlying motivation for MDD is to improve productivity." Once again, the idea connected to efficiency of the production process – and, thus, with all production phases – is brought under scrutiny. Hence, these model-based process and, more specifically, model-driven development, are clearly aligned.

MDD is mainly concerned with the development of software systems as dictated by the Model-Driven Engineering (MDE), a broad systems engineering discipline where models are the central artifacts of development and "used to communicate design decisions and generate other design artifacts", (Milicev, 2009). According to Jones (2010) 20% of defects encountered on software have their origin traced back to requirement specification; and up to 35% of them are related to coding alone. Hence, the need for efficient software design and development approaches is justified. It is hoped that MBSE and the like represent important steps toward this end. Some view Model-Driven Architecture (MDA) as a way forward in this direction.

2.1. Model Driven Architecture

MDA is OMG's (Object Management Group) particular solution to the MDD approach and is based on standards set forth by OMG itself, (Thomas, 2004). Concerning the software development, Miller and Mukerji (2003) write that MDA "is an approach to using models in software development. [It] is another small step on the long road to turning our craft into an engineering discipline". Moreover, we read from Milicev (2009) that the building of complex software systems should adopt the same techniques employed in other engineering disciplines, in which models and modeling are amply exploited.

MDA corresponds to a set of guidelines aiding the process of taking software requirement specifications and structuring them as computer models; it builds upon OMG's Unified Modeling Language (UML) 2.0 and Object Constraint Language (OCL), (Whittle, 2006). The key concept of MDA is to separate those "things" that change rapidly from those "things" that do not. Things that change rapidly are those related to the underlying platform technology: connectivity, architectures, hardware platforms, sensor technology, operating environments and so on. These evolve in a much faster pace than those concerned with business functionality and application behavior logic, such as, problem domain semantics, relations among domain concepts.

The MDA development lifecycle is similar to the traditional development lifecycle, but emphasis is given to creation of semi-formal models, i.e., models that can be input to computer. There are two types of models: Platform-Independent Model (PIM), which are those containing the business logic and are independent of technological aspects, and the Platform-Specific Model (PSM), with business logic and technological aspects

modeled together. These models can be automatically transformed into one another while preserving semantics, usually transforming models from the higher abstract level to the lower implementation-level or transforming within the same level: e.g., PIM-to-PIM, PIM-to-PSM, PSM-to-code. Observe here, that these transformations support automatic code generation. Hence, this model-driven approach, as with MDD more generally, treats software development as a chain of semantic preserving transformations between successive models, starting from early development phases: from requirements to analysis, to design, to implementation, to deployment. This is not achievable with UML alone. Though plain UML is strong in modeling structural aspects, it is weak in modeling behavior and thus the UML/OCL combination of the MDA approach helps define precise and unambiguous PIMs.

One way to increase productivity is through software reuse. Reuse has been applied to a myriad of artifacts in software development, ranging from broad-scope development reuse applications such as product, process, technology, and experience to more development-specific artifacts such as software component, architecture and requirements, (Ferré, Vegas, 1999). Other important model-driven advantages include automatic code generation, verification and validation, automatic software documentation, or even efficient software requirements specifications management.

For example, one could customize different software versions for a single target platform using one high-level software model as source. Alternatively, one could automatically generate code by way of high-level software models – which captures the essential, non-platform specific implementation – and develop software across different platforms.

In an independent study carried out by a middleware company, a 35% increase in productivity was reported to be achieved due to the adoption of the MDA approach, (Whittle, 2006). Although MDA requires more time and effort during the design phase, some measure of payoff is obtained in the implementation phase of development. It is thought that typically an average between 50-90% of automatic generated code can be achieved (Whittle, 2006). It is believed that these efficiency measures will increase with software component reuse in other subsequent projects.

The Executable and Translatable UML, or simply xtUML, which partially implements the MDA model transformation approach, is reviewed in the next subsection.

2.2. Executable and Translatable UML

The Shlaer-Mellor method, introduced in 1988, is an object-oriented software development method; see (Shlaer, Mellor, 1996). This method is also known throughout the modeling community as Object-Oriented Systems Analysis or Object-Oriented Analysis. The method has evolved to produce what is currently known as the eXecutable and Translatable UML (xtUML), which is both a software development method and a highly abstract software language. xtUML is a UML profile and some authors also refer to this development method simply as Executable UML. xtUML is considered by some to be a "full-fledged programming language", refer to (Starr, 2014).

xtUML employs unambiguous semantics by way of an action language, called *Object Action Language* or simply OAL, that allows models to be executed and verified against requirements early in the development life-cycle. This action language is used within the model itself. Four main types of modeling elements are used during model development and are summarized as:

- Components, Package diagrams: for model organization;
- Class diagrams: for data structure;
- State Machines: for specifying model behavior;
- OAL: for action description.

Other modeling diagrams can be used in BridgePoint during the modeling process to help explain some behavior, even though these are not considered for model verification or code generation purposes.

The BridgePoint modeling tool implements xtUML and, among other tools, it is being used as a software development

research asset at Atech. BrigdePoint integrates a UML editor, a model *verifier* – model debugging – and a code generation functionality called model transformation by the *model compiler*. It was developed by Mentor Graphics and has been reported in the literature to be used in industry projects at SAAB (Wedin, 2010), Ericsson AB – Sweeden, the Australian Research Council (Stien, 2006) and others[4], and is endorsed by some in the academy as well: Australian National University (Flint, et. al. 2004), Chalmers University of Technology and University of Gothenburg (Burden, 2012, 2014).

A couple of modeling case studies are presented in Subsections 2.3 and 2.4.

2.3. The ATC Case study

Details on comparison of some commercial modeling tools may be found in (Souza, Aguchiku, Gonzalez, 2015). In order to compare modeling tools in that study, a benchmark problem was devised to better understand tool modeling capabilities toward the goal of obtaining the same – or very similar, at least – target model. This benchmark problem consisted of a very simplified version of the Flight Data Processing (FDP) system present in current Air Traffic Control (ATC) systems which are deployed throughout the many ATC centers located in various parts in Brazil. The intended FDP model version of the real system is limited in scope but it should consist of software components able to keep track of the main states associated to actual (aircraft) flight plan changes during its life- cycle.

2.3.1. The Benchmark Problem

The Flight Data Processing system considered is basically a system that oversees and keeps historical and current record of all controlled, civilian manned flights inside the corresponding volume over a prescribed territorial region. All controlled flights in this volume of interest are meticulously managed following a strict set of protocols intended for safety and performance. For this goal, the FDP creates a *flight plan* or a computer representation of where and when each aircraft will fly in this volume of interest. Flight plan evolution is performed according to what is called flight plan control states: Inactive, Pre-Active, Active, Terminated, and Archived. In particular, the Active state is further classified by an additional set of states, or sub-states. These sub-states detail flight control *handover* operations between different control sectors, controllers' consoles, and even air-traffic control centers. An extensive set of flight markers or flight configuration parameters related to the flight plan is created in the FDP system and continuously updated. Some of these deal with how and when the system triggers flight plan stage evolution specifically. Different types of flight plans are defined in the FDP system. The FDP system continually receives messages from an external entity, or from human intervention. Some messages specify which type of flight plan should be created internally to represent every real, physical flight stance. Depending on the type of a recently created flight plan, flight states follow a particular evolution pattern with its own set of state-transition specification requirements.

The benchmark problem is defined as an FDP model that entails a flight state (and sub-state) managing system implementation for some different flight plan types. Messages to the FDP system model should be internally created in the model or received from external model user (operator) interface. From the brief description above for the FDP system, one is able to identify many software modeling elements: structural aspects of the model, such as components and classes (attributes, methods); behavioral aspects, including state-machines and messages; and actions.

2.3.2. System Modeling

System modeling requires a shift in the developer's way of thinking and many good practice modeling recommendations are available, see (Starr, 1999) for example. Additionally, as with the Object Oriented design approach, MDD demands more effort

[4] Visit xtUML.org

into modeling in early stages of development, which can be offset by cost needed for implementation and testing.

This design consists of four main software elements: two elements to deal with message exchange – a Generator and a Receiver, an element to represent the concept of Flight Plans and a Flight Plan Manager that will manage flight plans. The communication between these elements has been modeled via ports, using asynchronous messages, also known as signals.

The *Message Generator* can send either a flight plan creation message or a command message. The later changes the state of the lifecycle of a flight plan. The message sent by the *Message Generator* is received by the *Message Receiver* in one of its ports. After the reception of the signal, the *Message Receiver* interprets the message and sends a message (related to the signal received) to the *Manager*. The *Manager* interprets the message and deals with it by creating a new instance of a *Flight Plan*, or by sending a command to an instance of an existing flight plan. These actions have been mostly modeled on statecharts from the pool of elements. Two different simulations were performed with each modeling tool: an automatic test and a human-in-the-loop test. The human-in-the-loop test configuration was also used to assess how the model could interact with non-modeled elements (like legacy code).

In BridgePoint, the proposed software elements were modeled as classes. The *Manager* and *Flight Plan* were related in a one-to-many association and were placed in a single component (that will be referenced as the *Manager* component), while the *Message Generator* and the *Message Receiver* were placed in different components (*Generator* and *Receiver* components, respectively, Fig. 1), since asynchronous message exchange may be done only between components. The components communicate to each other via ports and interfaces.

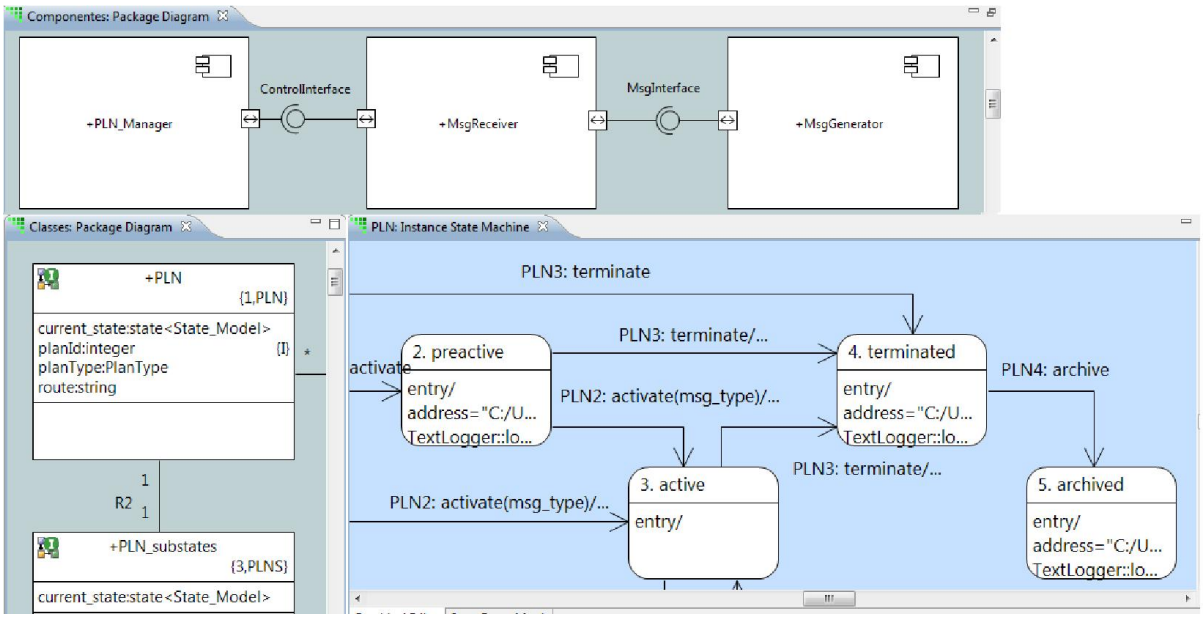


Figure 1: FDP system model components (upper view). The *Manager* component contains State (*PLN*) and Sub-State (*PLN_substates*) Classes. The Sub-State statechart is displayed in the bottom, right view.

Two interfaces were created, one containing the messages exchanged between the *Manager* and the *Receiver* components, and the other between the *Receiver* and the *Generator*. The actions were modeled in the classes statecharts, and were coded using Object Action Language (OAL), the action language used by BridgePoint. The resulting model was compiled, generating C code which then resulted in an executable file – on BridgePoint this is a single process.

Model interfaces define signals, the direction of messages and parameters that those signals carry. Inside some classes, the behavior of their instances was designed using statecharts. It is possible to create a class statechart, that define the behavior of the class itself, or an instance statechart, that defines the behavior of each one of the classes' instances. Both statecharts are not able to represent sub-states, being necessary to create a different class with a statechart to represent the possible sub-states of a state. The transitions on the statechart representing the flight plan sub-states were modeled to be managed by its parent statechart.

2.3.3. Model Verification

The general idea for model execution/ debugging is as follows: the Generator sends a signal (asynchronous message) to the Receiver. The Receiver decodes the signal, and sends another message to the Manager, informing which action should occur (creation, canceling, activation or other) and which flight plan the message is destined to. When the action occurs (the target flight plan is affected), the flight plan logs the current time and a brief description of the action in a file that is hardcoded in the model. An example of this description could be "Flight Plan with Id: 5 transitioned to pre-active state". This logging functionality is not originally part of BridgePoint and, therefore, it should be coded externally in a third party environment and then added to the model as an External Entity. As mentioned previously, two configurations were simulated: an automatic test and a human-in-the-loop test. The automatic test was done by use of OAL to generate the messages in the generator's statechart. The human-in-the-loop test was done by way of a Java user interface, devised specifically for model execution. This UI was introduced in the model as a "realized component", replacing the *Generator* component entirely. The execution was carried out in BridgePoint using the Verifier tool. Under this tool, the external code used (i.e., the code regarding the External Entity and Realized Component elements) was coded in Java language. If it were executed as a standalone application, it is assumed that this code would have to be manually integrated to the generated code or operated asynchronously as a distinct application; this standalone execution has not been yet performed.

2.3.4. Code Generation

Code generation is a one-step model-to-text transformation process in BridgePoint. Model diagrams and OAL are translated into target code language without creating an intermediate platform specific model (PSM). Legacy code written on the same target language, when available, can be attached together with model generated code.

In this regard, BridgePoint is not fully MDA compliant. Platform specificities are introduced into the development through the model compiler; persistence, multi-tasking, distributed computing and implementation of data structures are not dealt with OAL. These implementation issues are considered by the model compiler through *markings*, or the definition of some hardware and software architectural decisions. It generates code from models and was conceived to possess a modular structure, thus allowing customization to extend its original functionalities. Reuse of the software architecture is achieved by employing the same model compiler configurations to other application developments. A binary executable is also created by compiling code following this same code generation process.

2.4. ADS-B Tracking Service

A second model-driven related implementation case study consists in publishing flight Automatic Dependent Surveillance – Broadcast, or ADS-B, data in a Data Distribution Service (DDS) network.

ADS-B data is received and decoded with a low-cost, in-house hardware implementation solution consisting of an antenna and a portable pocket-sized computer. A dedicated software was developed with a modeling tool for publishing ADS-B derived tracks in a DDS bus network which are then consumed by other workstations belonging to this same network. The ADS-B data is used to track aircraft positions in a 3D virtual Geographic Information System (GIS) environment, also under development at Atech.

3. FORMAL SPECIFICATION

The verification by formal methods, and model checking in particular, is also an important pursued study objective. Research is being carried out in order to investigate the use of formal specification models and to perform systematic exhaustive checking of these models, or model checking, in contrast to the deductive verification or the theorem proving alternative, the other great approach category under the field of formal verification methods. Two case studies are presented next.

3.1. First Case Study: CSP modeling

This first study is a simplified version of the benchmark problem presented in Section 2. Since the benchmark problem involves message exchange between different components, a study is being carried out by specifying these system components using Communicating Sequential Process, or CSP, (Hoare, 1985). CSP is a process algebra that may be used as a formal specification language for describing interactions in concurrent systems. As a specification language, elements (mostly components) of a system are modeled as processes in CSP. Processes are seen as black boxes and it is possible to observe only what happens on the process interface. These observable elements are called events. Events may represent atomic actions or communication channels, in which processes may synchronize actions and message exchange through a channel. CSP has different semantics to express the meaning of processes and the one that will be focused in this section is the denotational semantics. The denotational semantics uses sets of behaviors to describe processes; for example, the Traces Model describes a process with the set of all possible traces of the process.

This case study employs FDR3, a recent replacement version for the Failures and Divergence Refinement tool, (Gibson- Robinson et. al., 2014) in order to verify the specified system. The main goals of this case study are to assess if and how CSP may be used to specify Air Traffic Control systems and what properties of ATC systems may be verified using FDR.

The FDR tool is a model checker, more specifically, a refinement checker. In CSP, the basic idea of refinement is a relation between processes; it is said that a process A is refined by a process B if everything allowed in process B is allowed by process A. This refinement relation is always related to a denotational semantics model. For example, the process A is refined by process B with respect to traces (Traces Model) if all possible traces of process B are also possible for process A. Besides refinement checking, FDR can check presence of deadlock, divergence and non-determinism.

The specified system is a simpler version of the system presented in Section 2.3.1. Flight plan control states are the same, though sub-states are not considered in this initial assessment for simplicity. Flight plan evolution is also simplified: every flight plan is created on the Inactive state; after receiving a pre-activation message, an internal process decides if the flight plan evolves to the Pre-Active state or if it remains on the Inactive state. While in the Inactive state, the flight plan only acknowledges the pre-activation message and ignores all others. This same procedure is adopted correspondingly for the other states: the system receives a message, if the message is the one that orders the flight plan to evolve to the next state, an internal process decides if the flight plan evolves to the next state. State transition sequence is ordered as follows: Inactive, Pre-Active, Active, Terminated and Archived. When the flight plan reaches the Archived state, it can no longer evolve and, thus, ignores any additional message.

The system high-level structure is the same: a message Generator sends messages to a Receiver; the Receiver decodes the message and passes the information to a Manager that deals with flight plan creation and evolution.

The Generator, Receiver, Manager and each flight plan were modeled as single processes in the CSP specification. Since the communication between the Generator and the Receiver is asynchronous, a buffer was included and modeled as a process emulating the use of asynchronous message exchange. The communication between Receiver and Manager and between Manager and a flight plan is synchronous. The Generator sends a message with an instruction (either *create*, *pre-activate*, *activate*, *terminate*, *archive*) and a flight plan id number to the buffer using channel "in". The Receiver consumes messages from the buffer, through channel "out" and sends a message with the flight plan id number using the channel corresponding to the message instruction, e.g., if a creation message is consumed, the Receiver will send a message using the creation channel. The Manager receives the message and then proceeds sending a message to the specified flight plan, sending a message using the channel corresponding to the instruction. A communication diagram (generated on FDR) of the system may be seen in Fig. 2.

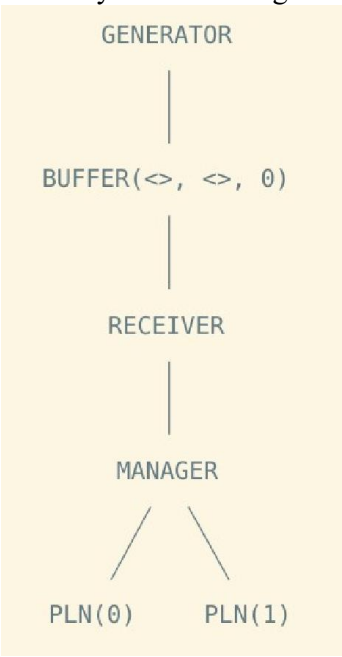


Figure 2: Tool generated communication diagram based on FDP system CSP modeling.

In this case study, an investigation was conducted in order to determine what could be verified with FDR. Using refinement checking, it is possible to check if the specification defines a system behavior that it must possess, or if the specification does not contain a behavior that the system must not present. For example, it is possible to verify if a specification has a determined trace by hecking if a process with only that trace refines the specification. If it refines the specification, it means that the trace is possible for the specification. In this case study, it was verified whether the Manager sent a message to a flight plan after receiving a set of messages. By exploiting other denotational semantics models with the FDR tool – mainly, Failures and Divergence Model we expect to further expand the quantity and improve the quality of the verification trials under consideration. The use of asynchronous communication through the buffering process described above introduced some concurrency to the specification. Using FDR it was possible to verify that the system was deadlock free (though this result was expected, since the processes of the specification were simple).

The verification on FDR was responsible for some changes on the specification of the system. Like most model checkers, FDR presents the state explosion problem (Clarke et. al., 2012). Because of this problem, it was feasible to verify the system with only two Flight Plan processes and by limiting the number of buffer messages to three messages. Initially, it was intended to employ an asynchronous communication between the Receiver and the Manager with an additional message buffer; though this was later changed to a synchronous communication. The verification was performed on a CentOS 6.7 workstation with Intel Core i5 (Quad-Core, 2.50GHz) and 8GB of RAM.

3.2. Second Case study: SMV modeling

The second case study makes use of the AutoFocus3 tool[5] (Hölzl & Feilkas, 2007), which is an open source modeling tool, built over the Eclipse platform. This tool implements a broad set of functionality, ranging from Requirements Engineering, Modeling and Simulation, Code Generation and Deployment to Testing / Formal Verification. It integrates the NuSMV tool (Cimatti et. al., 2002), a symbolic model checker jointly developed by a few groups in

[5] Visit http://af3.fortiss.org/

the academia[6] which evolved from the SMV (McMillan, 1993). The NuSMV tool performs formal verification on models by analyzing them as synchronous finite-state systems in the form of an automaton, that is, a representation of a formal language. One can express properties using Computation Tree Logic (CTL) or Linear Temporal Logic (LTL) and produce a formula that will be used by NuSMV to formally verify the model for those properties.

The methodology of the present subsection consisted in designing the architecture, simulating the exchange (sending) of messages and applying the model checking techniques via NuSMV. This technique consists in (model) checking a CTL/LTL property. If the property holds, a SUCCESS status is received. If it does not hold, a FAILURE status is received and a counterexample is generated.

As in the case study of Section 2, the case study described in this section was also based on the FDP requirements of a real ATC system implementation.

The modeling problem is described as follows: control of a flight must be transferred from one ATC sector/center to another. An instance of the flight plan exists in each of the two sector/centers involved. A flight plan in this specific scenario can be in *active*, *not active* or *in transference* status. Flight plans in the *active* status of two adjacent sectors at the same time are forbidden. Flight control of an aircraft will eventually be passed along to the next sector/center through a procedure known as *handover*. Though not a system required specification, handover will be modeled to be accomplished within ten time units. The receiving sector first acknowledges control handover as *in transference* and then communicates acceptance of control of that flight. The communication is made via messages, such as EST, ACP, and HND, which are exchanged by flight controllers.

A description of the Flight Plan State Machine, or FPSM, is presented next. FPSM-1 will designate the flight plan instance of the sector giving control over a flight; likewise, FPSM-2 will represent the flight plan instance of the receiver of flight control, Fig. 3. Both FPSMs start in the *Preactive* state. When the system receives an EST message, it transitions to *Active not controlled* state for both FPSMs. When it receives an ACP message, it transitions to *Active controlled* state in FPSM-1 and it starts a countdown timer from ten time units, which is a modeling feature used in the model for demonstration purposes. When it receives a HND message or when the timer reaches three time units, FPSM-1 transitions to *Active transference proposition*. An ACP message after that takes FPSM-1 to *Active not controlled* and FPSM-2 to *Active controlled*. If it receives an ACP message within the remaining three time units, FPSM-1 transitions back to *Active not controlled* and FPSM-2 to *Active controlled*, whereas if it does not, a timeout signal is generated and FPSM-1 transitions back to the *Active controlled* state.

This system was modeled using the AutoFocus 3 tool with the purpose of model checking it for safety properties with the NuSMV tool. Safety properties resemble undesired behavior which should never happen in a distributed system. The two safety properties to be verified in this study are given as follows:

1) It is forbidden that more than one flight plan instance is active at the same time;

2) When FPSM-1 is *active*, FPSM-2 is *in transference* and an ACP message is received, synchronization must occur so that FPSM-1 is not active and FPSM-2 is active.

The first version of the architecture was modeled with two FPSMs and a GUI panel to send the messages during Simulation. The messages sent are, in order: “EST Sync” to both FPSMs, ACP to one of the FPSM, HND to the same FPSM and “ACP Sync” to both FPSM.

The verification analysis showed that there were no unreachable states in any of the two state machines considered in this model; an expected outcome due to model construction.

Through simulation, it was noticed that an additional model component, named Controller State Machine (CSM), was needed to check if the flight plan was active in any of

[6] Visit http://nusmv.fbk.eu/

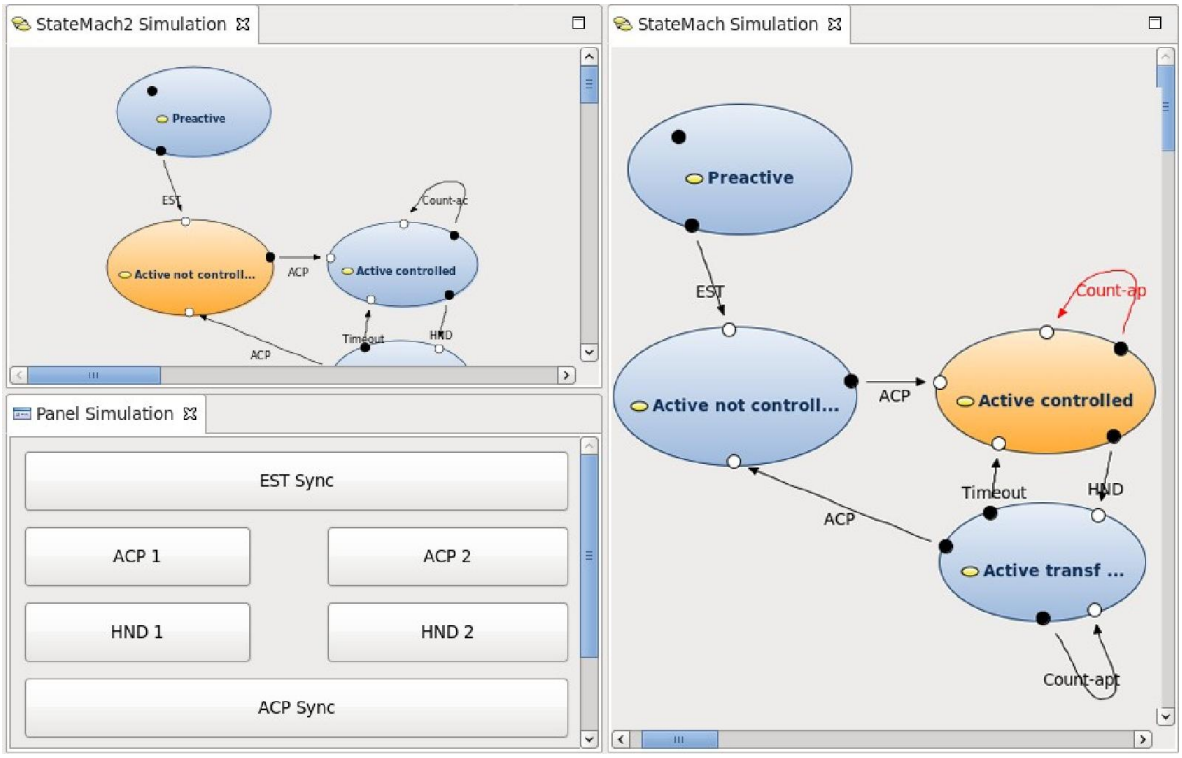


Figure 3: Implementation of two State Machines in AutoFocus 3 used for flight control transferring simulation between two ATC sectors. The user Panel, in the lower left corner, contains buttons for user input during simulation.

the two FPSMs, so that the ACP message would be ignored and property 1) could be assured. Property 2) was successfully checked.

The main objective of the Controller is to ensure the system works as expected and, therefore, it has to filter the messages it receives. A description of the CSM is presented next. It contains three states: *Not Controlled*, *Controlled* and *InTransference*. The Controller starts on *Not Controlled* state and persists this state, unless it detects that one of the flight plan instances status is *Controlled*. It also filters the initial messages: it only forwards the ACP message if it is sent by one instance only. When in *Controlled* state, it transitions to *InTransference* state only if it receives a HND message from a controlled instance or if it detects the flight plan instance status changed to *InTransference*. Then, it forwards the HND message. When on *InTransference* state, it forwards any message it receives and if none of the flight plan instance status is *InTransference*, it transitions back to *Controlled* state.

The second version of the architecture is shown in Fig. 4 and includes the CSM, detailed in Fig. 5, so that properties 1) and 2) can be verified, see Fig. 6. As stated previously, the counterexample analysis, when generated, helped identify possible flaws that could then be corrected and contributed for improving the overall system model architecture.

Analysis with the AutoFocus3 tool showed that unreachable states were not to be found for any of the two state machines; this was expected by their construction.

4. DISCUSSION

Although the use of MDD techniques and tools are apparently widespread in the academia and in industry, it is not unanimity. There are many who do not advocate support for them as the solution to systems and software development woes and hurdles; see (Den Haan, 2008; 2009). This conundrum is by no means simply polarized into a dichotomy about the effectiveness of MDD. Other proponents have come forward and

have contributed with additional viewpoints, such as agile methods and encouragement of use of domain specific languages to name a couple, all motivated with the general concern that systems development should be a more structured activity targeting an increase of efficiency. Whittle, Hutchinson, Rouncefield (2014), commenting on analyses based on a recent questionnaire research about adoption of model-driven engineering techniques in industry, claim: "Some companies have reported great success with it, whereas others have failed horribly." Some believe that failure related to model-driven techniques has to do with UML itself. M. Fowler (2003), for one, emphasizes that UML and other OMG standards are the platform used for development, the opposite of what MDA boasts about being able to provide for the development lifecycle: platform independence. He further argues that UML does not enable the sought for "jump" of abstraction level that many times it is praised to be able to offer for software development.

Though xtUML was based on UML2.0, and this was devised to have real time applications in mind, it seems that the industry has not accepted UML 2.0, a much anticipated major revision of the UML standard, the way it was intended in the first place by not reflecting "the literature on empirical studies of software modeling or software design studies", (Whittle, et.al., 2014). This seems to corroborate a claim by D. Thomas (2003) that UML 2 is too complicated and that one should "seek simpler solutions to adding yet another layer of meta-stuff", when referring to MDA's adopted MOF. Strong UML adherents, like B. Selic (2010) and S. Mellor (2006), oppose this view and defend that UML2 deserves no criticism when employed for real-time and even embedded applications. Other standards, however, such as MARTE, QoS-FT (Quality of Service – Fault Tolerant), fUML, UML profiles for security, etc. have been defined to circumvent industry perceived UML shortcomings.

Indeed, the functionality of code generation from models is overestimated and we have yet to find or implement design examples that handle well fault treatment scenarios and concurrency situations in complex, distributed systems. The quality of automatically generated code remains to be analyzed more carefully. Nevertheless, we understand some important gain from model driven development in systems (software) engineering is evident in terms of reuse, documentation, and maintenance.

The Lockheed Martin experience with MDD has produced interesting recommendations noteworthy of mention (Schmidt, 2006):

- Avoiding a one-language-does-all approach by exploiting multiple Domain Specific Languages in project development for narrow, well-understood domains;
- Automated generation of partial implementation artifacts;
- Integration of legacy assets through reverse engineering;
- Model verification and checking.

These surely decrease the designer's or developer's expectations with respect to MDD automatically generated artifacts. Hutchinson et.al. (2014) agree with the above when they state that code generation solely should not drive MDE adoption.

Some measure of moderation concerning the adoption of MDA tools is advised: "However, used in moderation and where appropriate, UML and MDA code generators are useful tools, although not the panaceas that some would have us believe", (Thomas, 2004).

5. CONCLUDING REMARKS

This preliminary note on software modeling is a partial spin-off of the ongoing research effort into evaluation of model-based methods and tools for software production.

It is expected that this effort will lead to a mapping of methods, modeling tools to work with in order to achieve the goals set forth by the MBSE approach and apply them to in-house Software Engineering.

Two types of simulations were presented in order to analyze the system specification and model. Model checking is applied to verify formal specifications of a system, a verification process which is

performed automatically. Analysis on verification results is intuitive and, hence, this type of simulation is simple once what is to be verified is clearly defined. The other simulation discussed was model execution

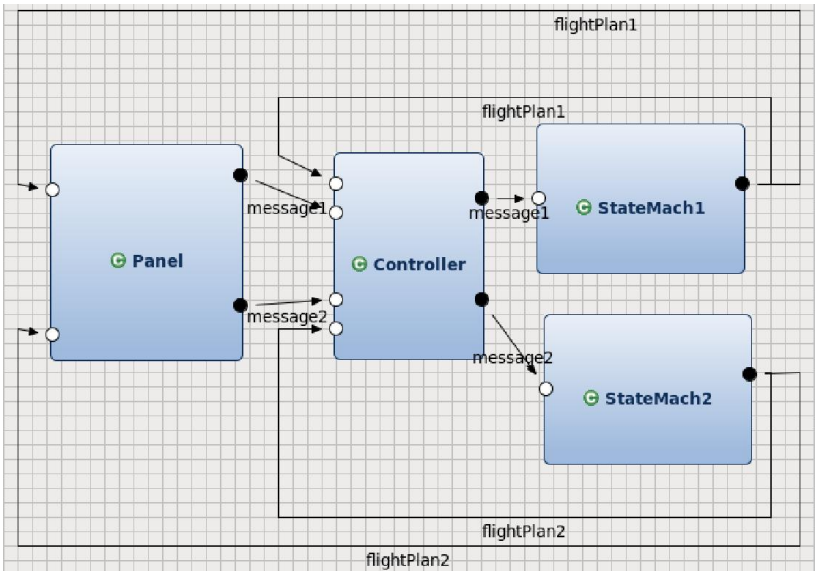


Figure 4: FDP system model architecture in AutoFocus 3.

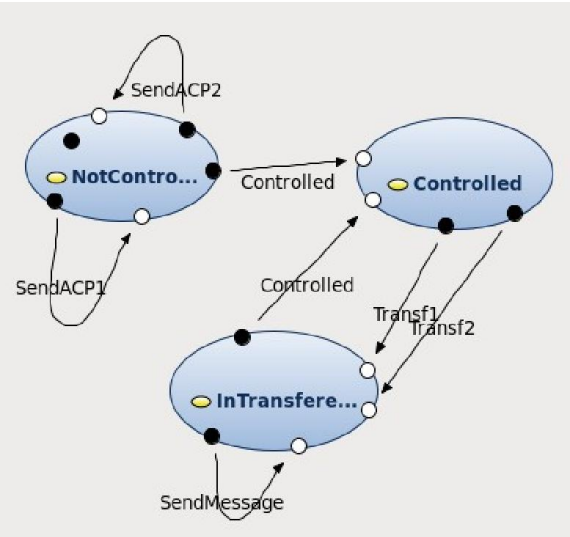


Figure 5: Modeling for the Controller State Machine (CSM).

Specification Atom	Verification c	Last check date	Last check result
Property 'planoVooIn1 == Controlado( ) && planoVoc	Arch_v1	Wed Aug 26 11:25	SUCCESS
If '((planoVooIn1 == Controlado( ) && planoVooIn2 =	Arch_v1	Wed Aug 26 11:26	SUCCESS

Figure 6: Success results for the verification of properties 1) and 2).

employed to verify a model of a system. Model execution is inherently a more laborious process, since every – or the majority of it – possible scenarios should be manually tested; either by coding these scenarios in the model or by human-in-the-loop simulations. This simulation, however, represents a development artifact closer to the implementation, since code may be generated directly from the model itself, while model checking verifies if the problem is well specified. Even if a system is well specified,

the resulting system may still contain errors, given the gap between the specification and the system; whereas the gap between the model and the system is ideally smaller, resulting in fewer differences between them. This discussion reflects the main difference between formal specification and system modeling. It is expected that further studies and research will decrease the gap between specification and model.

"Good design, sound architecture and common sense play as much a role in development using" model-driven methods as in traditional style approach; see (Den Haan, 2009)[7]. Raising the level of abstraction for development is what MBSE is about. Although mistakes introduced at a higher level of abstraction can impact the project as a whole, MBSE can help detect mistakes in early development phases and even help avoiding some mistakes from occurring at all. Working with models – however simple they may be and yet complete versions of business logic – is more convenient for analysis and verification through model execution or formal methods, during product development phases, and also for product maintenance phase after product deployment. This is mainly due to the ease of understanding of a well-structured model construction and by facilitating communication amongst team members.

We close this partial presentation on the issue with important comments from Frank Truyen. According to Truyen (2006), the transition from a traditional software development approach to a model driven one is a major overtaking, an effort requiring "formal planning, orchestration and sponsorship." Moreover, Truyen states that the this transition process demands thorough planning for its implementation over a period of many years, a detailed management of resources, and subsequent continuous monitoring of activities results. He mentions that this model driven shift should be supported and funded by management. "Moving too fast, out-of-sequence, or proceeding without proper planning simply puts the whole transition process at risk. The main ingredient for ensured success is keeping the end-vision of the transition in sight." (Ibid)

[7] Contributed by Vinay Kulkarni on Aug 07, 2008 08:05.

6. REFERENCES

ATKINSON, C., KÜHNE, T. "Model-Driven Development: A Metamodeling Foundation". IEEE Software, 2003. September/October. p.36-41.

BASILI, V. R., CALDIERA, G. "A reference architecture for the component factory". ACM Transactions on Software Engineering and Methodology, 1992. v.1 No. 1. p. 53-80.

BURDEN, H. Three Studies on Model Transformations - Parsing, Generation and Ease of Use. 2012. Thesis for the Degree of Licentiate of Philosophy. Chalmers University of Technology and University of Gothenburg, Also: A Scholarship Approach to Model-Driven Engineering, Chalmers University of Technology, 2014, PhD dissertation.

CAPRIO, G. "The software factory: Making the most of software reuse". 2008. Available at: <http://searchwindevelopment.techtarget.com/tip/The-Software-Factory-Refactoring-an-industry>. Accessed on 15 Apr. 2014.

CARROZZA, G. et al. "Engineering Air Traffic Control Systems with a Model-Driven Approach". IEEE Software, 2013, May/June, 42-48.

CERNOSEK, G., NAIBURG, E. "The Value of Modeling". White paper, IBM/Rational Software, 2004, June.

CIMATTI, A., et. al. "NuSMV 2: An OpenSource Tool for Symbolic Model Checking". Proceedings of the 14th International Conference on Computer Aided Verification (CAV '02), 2002. Ed Brinksma and Kim Guldstrand Larsen (Eds.), Springer-Verlag, London, UK, UK, p. 359-364.

CLARKE, E. M. et. al., "Model checking and the state explosion problem". *Tools for Practical Software Verification, 2012. Springer Berlin Heidelberg.* p. 1-30.

DEN HAAN, J. D. "Eight reasons why Model-Driven Development is dangerous". June 27, 2009. Available at: <http://www.theenterprisearchitect.eu/blog/2009/06/25/8-reasons-why-model-driven-development-is-dangerous/>. Accessed on 28 Aug. 2015.

DEN HAAN, J. D. "Eight Reasons Why Model-Driven Approaches (will) Fail". July 28, 2008. Available at: <http://www.infoq.com/articles/8-reasons-why-MDE-fails#Atk02>. Accessed on 16 Apr. 2014.

FERRÉ, X., VEGAS, S. "An Evaluation of Domain Analysis Methods". 4th CAiSE/IFIP8.1 International Workshop in Evaluation of Modeling Methods in Systems Analysis and Design - EMMSAD99, 1999.

FLEMING, C. H., LEVESON, N. G., PLACKE, S., "Assuring Safety of NextGen Procedures", Tenth USA/Europe Air Traffic Management Research and Development Seminar (ATM2013), 2013.

FLINT, S., GARDNER, H., BOUGHTON, C. "Executable/Translatable UML in Computing Education". Proceedings of the Sixth Australasian Conference on Computing Education, 2004, v. 30.

FOWLER, M. "Platform Independent Malapropism". 2003. Available at: <http://martinfowler.com/bliki/PlatformIndependentMalapropism.html>. Accessed on 26 June 2015.

GIBSON-ROBINSON, T. et al. "FDR3—A modern refinement checker for CSP". *Tools and Algorithms for the Construction and Analysis of Systems*, 2014. Springer Berlin Heidelberg. p. 187-201.

HOARE, C. A. R. "*Communicating Sequential Processes*". Prentice-Hall, Inc., 1985, Upper Saddle River, NJ, USA.

HÖLZL, F., FEILKAS, M. "AutoFocus 3: a scientific tool prototype for model-based development of component-based,

reactive, distributed systems". Proceedings of the 2007 International Dagstuhl conference on Model-based engineering of embedded real-time systems, 2007 Springer-Verlag, Berlin, Heidelberg, pp. 317-322.
HUTCHINSON, J., WHITTLE, J., ROUNCEFIELD, M. "Model-driven engineering practices in industry: Social, organizational and managerial factors that lead to success or failure". Science of Computer Programming, 2014. v.89. p. 141-161.
INCOSE. International Council on Systems Engineering, Systems Engineering Vision 2020, INCOSE – TP – 2004 – 004 – 02, Version 2.03, September 2007, Available at: <http://oldsite.incose.org/ProductsPubs/pdf/SEVision2020_20071003_v2_03.pdf>. Accessed on 28 Aug. 2015.
JONES, C., "Software Quality and Software Economics", Software Technology News, Vol.13, n.1, April, 2010, p.10-14.
MCMILLAN, K. "Symbolic Model Checking". Kluwer Academic Publishers, 1993. Norwell, MA, USA.
MILICEV, D. "Model-Driven Development with Executable UML". Wiley Publishing, Inc., 2009.
MELLOR, S. "Demystifying UML". 2006. Available at <http://www.eetindia.co.in/ARTICLES/2006MAR/PDF/EEIOL_2006MAR01_EMS_TA.pdfSOURCES=DOWNLOAD>. Accessed on 25 Apr. 2014.
MILLER, J., MUKERJI, J. (editors). "MDA Guide Version 1.0.1". Object Management Group, June 12, 2003. Doc. No. OMG/2003-06-01, Available at: <http://staffwww.dcs.shef.ac.uk/people/A.Simons/remodel/papers/MDAGuide101Jun03.pdf>. Accessed on 28 Aug. 2015.
PORTIER, B., ACKERMAN, L. "Model Driven Development Misperceptions and Challenges", January 21, 2009, Available at: <http://www.infoq.com/articles/mdd-misperceptions-challenges>. Accessed on: 16 Apr. 2014.
SCHMIDT, D. C. "Model-Driven Engineering". IEEE Computer, 2006. February. p. 25-31.
SELIC, B. "UML2: Designed for Architects", IEEE Software, 2010. Nov/Dec.
SHLAER, S., MELLOR, S. J., "The Shlaer-Mellor Method", Project Technology, 1996. Available at: <http://ooatool.com/docs/SMMethod96.pdf>, Accessed: 19 Oct. 2014
SOUZA, E. C., AGUCHIKU, F. S., GONZALEZ, A. L. "A Comparison of Selected Tools for The Model Based Engineering Approach to Complex Software Development". Seminário Embraer de Tecnologia e Inovação – SETI, June, 2015, São José dos Campos.
STARR, L., "xtUML.org website. xtUML - eXecutable Translatable UML Open Source Editor". Online: <https://www.xtuml.org>. Accessed on 9 Apr. 2014.
STIEN, M. "Executable Translatable UML for Enterprise Applications", 2006. Available at: <http://www.softimp.com.au/Common%20content/White%20Papers/xtUML%20for%20Enterprise%20Applications.pdf>. Accessed on 28 Aug. 2015.
THOMAS, D. "MDA: Revenge of the Modelers or UML Utopia?". IEEE Software, 2004. May/June. p.15-17.
THOMAS, D. "UML – Unified or Universal Modeling Language?", Journal of Object Technology, v.2, no. 1, Jan/Fev, 2003, pp.7-12
TRUYEN, F. "The Fast Guide to Model Driven Architecture – The Basics of Model Driven Architecture". White paper, Cephas Consulting Corp., 2006, January.
VÖLTER, M. "Architecture as a Language". IEEE Software, 2010. March/April. p. 56-64.
WEDIN, E. "Applying Model-Driven Architecture and SPARK ADA: A SPARK ADA Model Compiler for xtUML". 15th International Conference on Reliable Software Technologies, 2010. SAAB Bofors Dynamics AB, Sweden.
WHITTLE, J., KWAN, R., SABOO, J. "From scenarios to code: An air traffic control case study". Software and System Modeling, 2005. v. 4. p. 71-93.
WHITTLE, J., "Model-driven software development: What it can and cannot do". Information & Software Engineering, George Mason University, slide presentation, 2006. Online: <http://ewh.ieee.org/r2/wash_nova/computer/archives/jun06.pdf>.
WHITTLE, J., HUTCHINSON, J., ROUNCEFIELD, M. "The State of Practice in Model-Driven Engineering", IEEE SOFTWARE, May/June, 2014, p. 79-85

7. LIST OF ACRONYMS

3D	Three-Dimensional
ADS-B	Automatic Dependent Surveillance – Broadcast
ATC	Air Traffic Control
ATM	Air Traffic Management
CSM	Controller State Machine
CSP	Communicating Sequential Process
CTAS	Center TRACON Automation System
CTL	Computation Tree Logic
DDS	Data Distribution Service
FDR	Failures and Divergence Refinement
FDP	Flight Data Processing
FPSM	Flight Plan State Machine
INCOSE	International Council on Systems Engineering
MARTE	Modeling and Analysis of Real-Time and Embedded
MBSE	Model Based Systems Engineering
MDA	Model-Driven Architecture
MDD	Model-Driven Development
MDE	Model-Driven Engineering
MOF	Meta-Object Facility
LTL	Linear Temporal Logic
OAL	Object Action Language
OCL	Object Constraint Language
OMG	Object Management Group
PIM	Platform Independent Model
PSM	Platform Specific Model
SMV	Symbolic Model Verifier
UML	Unified Modeling Language
TRACON	Terminal Radar Approach Control
xtUML	Excutable, Translatable UML